CS2 Language Processing note 9

Top-Down and Bottom-Up Parsing

Recall that *parsing* is the activity of checking whether a given string of symbols (usually, the stream of tokens produced by a lexical analyser) is in the language of some grammar, and if so, constructing a parse tree for it.

[Actually, in practice, we would like a parser to do rather more than this. If the string is not in the language, a helpful parser should issue a sensible *error message* which helps the user to pinpoint the error. Furthermore, if the parser discovers one error somewhere in the string, it should ideally be able to somehow "recover" from this error and continue parsing, so that it can inform the user of all the errors in the input rather than just the first one. Good error reporting and error recovery are in fact quite hard to achieve, since a parser cannot in general know what the user intended. We will therefore leave these interesting issues to one side for now.]

There are two main kinds of parsers in use, named for the way they build the parse trees. A *top-down* parser attempts to construct a tree from the root, applying productions forward to expand nonterminals into strings of symbols. A *bottom-up* parser builds the tree starting with the leaves, using productions in reverse to identify strings of symbols that can be grouped together. In both cases the construction of the derivation is directed by scanning the input sequence from left to right, one symbol at a time.

An example

The following grammar is for a small language for describing electrical circuits:

$$C \rightarrow \textbf{seq}\ B \mid \textbf{par}\ B \mid \text{basic} \qquad\qquad B \rightarrow C\ B \mid \textbf{end}$$

The start symbol is C, for *circuit*; the other nonterminal B stands for a *block* of circuits ending with the keyword **end**. Here basic represents a token describing some basic component such as a resistor or capacitor. An expression of the form **seq** $C_1 \ldots C_n$ **end** is used to represent n components connected in series, while **par** $C_1 \ldots C_n$ **end** represents n components connected in parallel. We use '|' to join productions that share the same left-hand side, so for example $t \rightarrow u \mid v \mid w$ abbreviates three productions $t \rightarrow u$, $t \rightarrow v$ and $t \rightarrow w$. [Note that $t \rightarrow u \mid v \mid w$ itself is not a production, but an abbreviation for three productions.]

Consider now the following sentence:

seq par basic basic **end** basic **end**

To parse this from the *top down*, we begin with the start symbol C, and repeatedly choose a production from the grammar to expand a nonterminal. At each stage we look at the input to see which is the appropriate production to use. First, seeing that the sentence begins with **seq** rather than **par** or basic, we conclude that the first rule to apply must be $C \rightarrow$ **seq** B. We therefore have to parse the remainder of the sentence as a block B. Seeing that the second token in the string is *not* **end**, we conclude that the next rule to apply must be $B \rightarrow CB$. We have thus constructed the first two steps in a derivation:

$$C \Rightarrow \textbf{seq}\ B \Rightarrow \textbf{seq}\ C\ B$$

We now have to interpret some portion of the string as a circuit, starting after the initial **seq**. Seeing that the next token is **par**, we conclude that the rule to apply must be $C \rightarrow$ **par** B. The third step in the derivation is therefore

$$\textbf{seq}\ C\ B \Rightarrow \textbf{seq}\ \textbf{par}\ B\ B$$

We may continue in this way until we have a complete derivation for the sentence. [**Exercise:** Finish this off, constructing the corresponding parse tree as you go.]

Notice that this procedure produces a *leftmost* derivation — that is, one in which we always choose the leftmost nonterminal to expand. This is coupled with a left-to-right scan of the input string, in which at each stage we use the next token to help us decide which production to apply. Fortunately, for this grammar, the next token is always enough to tell us what the next production should be.

Whereas in top-down parsing we begin with a single "start symbol" and apply productions until we arrive at the whole sentence, in *bottom-up* parsing we start with the sentence itself and construct a derivation backwards until we arrive at the start symbol. We do this by scanning the input until we find a symbol or group of symbols that matches the *right hand* side of one of our productions. We then apply the production "backwards", replacing this group of symbols by the single nonterminal on the left of the production.

For example, scanning the above sentence from left to right, we discover that the third token matches the right hand side of the rule $C \rightarrow$ basic. We may therefore rewrite the sentence as follows:

$$\textbf{seq}\ \ \textbf{par}\ \text{basic basic}\ \textbf{end}\ \text{basic}\ \textbf{end} \Leftarrow \textbf{seq}\ \ \textbf{par}\ C\ \text{basic}\ \textbf{end}\ \text{basic}\ \textbf{end}$$

(We use the arrow $\Leftarrow$ here to indicate that this is a backwards derivation step.) Continuing for two more steps, we notice that the next two tokens basic and **end** can likewise be rewritten as C and B respectively, so we arrive at the form

$$\textbf{seq}\ \ \textbf{par}\ C\ C\ B\ \text{basic}\ \textbf{end}$$

But now we notice that the pattern $C\ B$ occurs as the right hand side of the rule $B \rightarrow CB$, so these two symbols may be grouped together and replaced by a B:

$$\textbf{seq}\ \ \textbf{par}\ C\ C\ B\ \text{basic}\ \textbf{end} \Leftarrow \textbf{seq}\ \ \textbf{par}\ C\ B\ \text{basic}\ \textbf{end}$$

We can continue in this way until the whole sentence collapses down to the single start symbol *C*. [**Exercise:** Finish this derivation, and draw the fragments of the parse tree that have been built up at each stage.]

Once again, this method uses a left-to-right scan of the input, and at each stage we perform a rewrite as far to the left as possible; and once again, for this grammar there only one possible choice of production at each stage. However, because we are constructing a derivation backwards, this means (if you think about it) that the derivation we end up with will be a *rightmost* derivation — that is, one in which it is always the rightmost nonterminal that is expanded. Indeed, it is typically the case that top-down parsers build leftmost derivations, while bottom-up parsers build rightmost derivations. Of course, the parse tree we get will be exactly the same either way (provided the grammar is unambiguous).

Both of these parsing strategies are used in practice, and neither of them is clearly the better. In general, top-down parsers are a little less powerful, but their algorithms are rather simpler to understand and to implement. For the rest of this thread we will concentrate on top-down parsing.

A trickier example

Here is another grammar, this time for a fragment of a programming language:

$$\begin{array}{lcl} \textit{stmt} & \longrightarrow & \mathsf{var} = \mathsf{var} \mid \textbf{if } \textit{cond} \textbf{ then } \textit{stmt} \mid \\ & & \textbf{if } \textit{cond} \textbf{ then } \textit{stmt} \textbf{ else } \textit{stmt} \\ \textit{cond} & \longrightarrow & \mathsf{var} == \mathsf{var} \end{array}$$

Here the start symbol is *stmt*, and the terminal var stands for a token corresponding to a program variable. Suppose we try to parse the statement

$$\textbf{if } \mathsf{var} == \mathsf{var} \textbf{ then } \mathsf{var} = \mathsf{var}$$

using the top-down procedure above. We begin with the symbol *stmt*, but what production should we apply first? If we are scanning the sentence from left to right and we only look at the "next" symbol (in this case the first token, **if**), there is no way to choose between the two productions

$$\textit{stmt} \longrightarrow \textbf{if } \textit{cond} \textbf{ then } \textit{stmt}, \qquad \textit{stmt} \longrightarrow \textbf{if } \textit{cond} \textbf{ then } \textit{stmt} \textbf{ else } \textit{stmt}$$

There are two ways to respond to this difficulty. We can either *look ahead* to see what comes further on in the sentence, or we can pick one production and *backtrack* if this later turns out to be the wrong choice. Both of these are expensive options, as they involve scanning parts of the input more than once. In the above example, by looking seven tokens ahead we would discover that we reached the end of the sentence without encountering an **else**, so the first of the above rules is the right one. But if the **then**-clause were some large and complicated statement, we would have to look ahead a very long way. Similarly with backtracking: if we picked the wrong rule at first, we could spend a great deal of time parsing the next part of the sentence before discovering that our first production was wrong after all.

Predictive parsing

It is far better if we can work with a grammar for which backtracking or lookahead is not required (such as the circuits example). Top-down parsing without backtracking or lookahead is known as *predictive parsing*. It only applies to certain context-free grammars. In order for a grammar to be suitable for predictive parsing it must always be possible to tell, given the current input token a and the nonterminal A to be expanded, which one of the productions for A generates a string beginning with a. There must be at most one such production in order to avoid the need for lookahead or backtracking. If there is no such production then no parse tree exists and the sentence is not in the language, so a parsing error is returned to the user. As we shall see, the information about which production to use in a given circumstance can be conveniently recorded in a *parse table* in which we can simply look up the production that is appropriate for a given a and A.

Another property that is needed for predictive parsing to work is that the grammar must not be *left-recursive*; i.e. there must be no nonterminal A such that $A \Rightarrow^* At$ for any string t. The problem with left-recursive grammars is that they give rise to infinite loops during parsing: an attempt to expand A may give rise to an infinite derivation $A \Rightarrow^* At \Rightarrow^* Att \Rightarrow^* \cdots$.

Predictive parsers are also known as *LL(1) parsers*. The first 'L' means they read input from the left; the second 'L' means they construct leftmost derivations (being top-down parsers); and the '1' means they look just one token ahead. Agrammar that is suitable for predictive parsing (i.e. one with the above two properties) is known as an *LL(1) grammar*. LL(1) parsers are the most widely used class of top-down parsers. There are also some commonly used classes of bottom-up parsers, such as LR(1) (because they build *rightmost* derivations) and the variant LALR ("lookahead LR"), but we will not study these in detail here.

Predictive parsing obviously works well when every construct of the language begins with a keyword such as **if**, **while** or **begin** which immediately identifies the construct that it is. In other cases, as we shall see, predictive parsing still works, but for not quite such obvious reasons. In still other cases, the grammar we are given is not suitable for predictive parsing but we can transform it into an *equivalent* one that is. And in yet other cases, neither the given grammar nor any equivalent one is suitable for predictive parsing. We will consider all these points in more detail in the remaining lectures.

Exercises. (1) Finish stepping through the examples of top-down and bottom-up parsing on page 2. (2) Using the grammar on page 3, what happens if you try to parse a statement of the form **if** *cond* **then if** *cond* **then** *stmt* **else** *stmt*? (3) Pick a small part of your favourite programming language, and write a context-free grammar for it. Would your grammar allow predictive parsing? Would an easy form of bottom-up parsing work?

John Longley 2002, Ian Stark 2001